Durable Hardware Transactional Memory for Extended Asynchronous DRAM Refresh Architectures

JOÃO PINHEIRO

Byte-addressable Persistent Memory (PM) technologies present a new paradigm for interacting with non-volatile memory, allowing applications to access PM directly and with much lower latency than before. Unfortunately, the combination of PM with Hardware Transactional Memory (HTM) has been far from trivial to implement due to the volatile nature of CPU caches, requiring the use of software instrumentation and techniques like Shadow Paging (SP) to guarantee durable HTM to PM. The commercial release of Intel Optane DC PM and the support for systems with Enhanced Asynchronous DRAM Refresh allows for CPU caches to be considered persistent as well, greatly simplifying the model for durable HTM. However, the use of software instrumentation techniques like Shadow Paging can still provide significant benefits for durable HTM solutions by taking advantage of the higher performance and lower latency of DRAM. This dissertation presents and evaluates a new solution based on the use of DRAM shadow paging for architectures with PM and durable CPU caches in order to improve performance and throughput.

Additional Key Words and Phrases: Persistent Memory, Transactional Memory, Shadow Paging, Extended Asynchronous DRAM Refresh, Hardware Transactional Memory

1 INTRODUCTION

Persistent storage options for computing systems have traditionally been limited to mass storage devices such as hard disk drives (HDD) and solid-state drives (SSD), whose performance is orders of magnitude slower than volatile main memory (DRAM). These mass storage devices are not directly accessible to applications and instead require the use of Application Programming Interfaces (API) provided by the Operating System (OS), along with costly serialization and deserialization processes to translate the data between its volatile representation and a format that can be stored on non-volatile devices.

The emergence of new byte-addressable Persistent Memory (PM) technologies like Intel Optane DC Persistent Memory opened the door to a new paradigm for interacting with non-volatile memory. These new PM technologies offer performance that is closer to DRAM and can be connected to the processor's memory bus.

The development of concurrent applications that can take advantage of these new PM technologies has led to significant attention in research into the implementation of Persistent Transactional Memory in systems equipped with PM and HTM.

However, HTM's reliance on volatile CPU caches means that committed transactions cannot be guaranteed to be atomically persisted to PM due to the possibility of remaining in the cache. It is thus required to complement transactions with complex software instrumentation in order to ensure durable HTM.

This approach of combining transactions with additional software instrumentation in order to guarantee atomicity and durability has been successfully utilized by state-of-the-art approaches such as NV-HTM[1], DudeTM[2], cc-HTM[3], Crafty[4], and SPHT[5].

Author's address: João Pinheiro, joao.pinheiro@tecnico.ulisboa.pt.

One technique that is particularly notable is the use of Shadow Paging (SP)[1; 2; 5] in combination with Write-Ahead Logging (WAL), which is used in all these solutions, with the exception of Crafty. With this technique updates are performed on local private copies rather than directly over the original data, allowing easier modification without the issue of consistency constraints. The original data is then replaced by the shadow copy, making the updates durable.

More recently, the introduction of new persistence domains for computing systems with small amounts of reserve power, such as Intel Extended Asynchronous DRAM Refresh (eADR), offered the possibility of treating CPU caches as non-volatile. In this new environment, HTM transactions that deal exclusively with PM data are able to rely entirely on hardware-level instructions without the need for any additional instrumentation in order to ensure durability.

However, despite offering performance that is significantly faster than persistent mass storage devices, current PM modules still have higher latency and slower write speeds than DRAM. It can thus be desirable to use an algorithm that makes use of DRAM in order to increase performance and reduce latency while taking advantage of HTM and the new eADR persistence domain for persistence[6].

Shadow Paging is a possible solution for achieving this goal. However, existing durable HTM solutions based on Shadow Paging have been designed for an ADR environment and are thus more computationally expensive than necessary in eADR. These solutions include mechanisms which are no longer required and do not take into consideration particular idiosyncracies of eADR environments, such as the possibility of flush operations being used to improve the performance of write operations.

2 BACKGROUND AND RELATED WORK

2.1 Persistent Memory

The advent of byte-addressable PM technologies has the potential to revolutionize the way in which data-intensive applications are developed. When compared to DRAM, these new technologies have significantly higher storage density, lower power consumption, and the ability to retain their contents for extended periods of time in the absence of power. Additionally, unlike traditional NVM mass storage devices, these new technologies are byte-addressable and connect directly to the computer's memory bus, allowing applications to address them directly without the need for translation steps to serialize and deserialize between representations or to go through any Operating System APIs.

However, despite the groundbreaking new features, these new persistent memory technologies also have notable drawbacks. Latency times for write operations are significantly higher than for read operations, which may cause serious performance degradation in write-intensive applications. There is finite write endurance, meaning that there is a limited number of times each bit may be

written before failure. And finally, bandwidth and performance are still limited compared to DRAM.

2.1.1 Intel Optane Persistent Memory. Intel Optane is the first, and currently the only, PM technology commercially available on the market. Optane DC memory is available in two different formats: as an NVMe SSD storage module that connects to the PCIe bus just like all other NVM mass storage devices, and in a DIMM format, which uses the physical DDR4 packaging and memory bus. It is this latter format, Optane DC, that is of most interest for PM applications, allowing it to be byte-addressable and much closer to DRAM in terms of latency. Optane DC PM offers a latency of up to about 350 ns, several orders of magnitude lower than the typical 10,000–100,000 ns latency of NAND-based SSD mass storage devices, but still higher than the 10–20 ns latency range for DDR4 DRAM[7].

However, despite the dramatic improvement in terms of latency, the bandwidth performance of Intel Optane, especially in terms of write operations, is still significantly worse than DRAM and closer to SSD mass storage devices[8].

Intel advertises a write endurance of 60 drive writes per day for an average lifetime of 5 years. While this level of endurance is significantly higher than the write endurance of NAND-based SSD mass storage devices, it can still become problematic when used as main memory in write-intensive applications[8].

2.1.2 Persistence Domains. One of the ideas that are central to computing systems with PM devices is the concept of a Persistence Domain (PD); the definition of the region of a computing system that is able to guarantee persistence. Once data reaches the PD, it can be guaranteed to have been persisted and recoverable upon system restart. The PD is not just limited to PM devices though; it may also include volatile devices that are able to hold state for long enough to be able so that it can be guaranteed to reach a persistent device. Intel Optane DC PM supports two different persistence domains, Asynchronous DRAM Refresh (ADR) and Enhanced Asynchronous DRAM Refresh (eADR).

In an ADR environment, the system has enough reserve energy to flush the memory controller's Write Pending Queue (WPQ) to PM. This means that it is sufficient for the application data to reach the memory controller in order to guarantee that it can be considered persistent under ADR.

In an eADR domain, computing systems have a higher amount of reserve energy than in ADR which, in addition to providing enough power to flush the memory controller's WPQ, also have enough power to allow the system to execute the required instructions to guarantee that CPU caches are also flushed to PM. The inclusion of the CPU caches in the eADR persistence domain brings significant advantages over ADR to the PM programming model. In an ADR environment data becomes visible to other cores via the CPU L3 cache layer before it has a chance to reach the memory controller and become persistent. It is also necessary for applications to explicitly flush caches in order for a store to become persistent. In an eADR environment, it is no longer necessary for applications to flush the caches in order to ensure persistence, and data becomes persistent before it becomes visible to other cores through the L3 cache layer. This offers the opportunity to simplify the programming model for durable HTM transactions by allowing durable HTM transactions to be executed entirely without the need for software instrumentation since values on the CPU cache can now be considered persistent.

2.2 Transactional Memory

The concept of a transaction was initially developed for database systems that needed to perform a set of operations that could manipulate data atomically in order to allow concurrent access to the data without sacrificing consistency. Database transactions guarantee this by ensuring 4 essential properties: atomicity, consistency, isolation, and durability (ACID)[9].

These properties are also essential in the context of parallel and concurrent programming where critical sections of code need to be executed sequentially and in isolation. This can be ensured through the use of locks, but lock-based programming is difficult to tune and notoriously known for being prone to programming errors[10; 11]. Global or coarse-grained locks are easier to implement, but can seriously degrade performance and restrict parallelism. Fine-grained locking allows for better parallelism and performance but is complex to implement and prone to errors that can be difficult to identify.

2.2.1 Hardware Transactional Memory. Transactional Memory implemented at the hardware level does not require the instrumentation of read and write operations. Instead, it relies on the cache coherence protocol to achieve atomicity and isolation without suffering from the high costs and overheads of instrumentation. This allows for better performance but also means that currently available HTM implementations are considered best-effort[12; 13]. Best-effort HTM is limited by cache capacity and cannot handle transactions that are too large to fit in the private CPU cache.

2.3 Combining Transactional Memory with Persistent Memory

The use of byte-addressable persistent memory technology enables applications to access and modify durable state without the need for an intermediate layer like the OS filesystem API, but still requires developers to ensure that modifications to the data are consistently applied to PM in the event of a failure. This is especially true when trying to guarantee the failure-atomicity of multiple operations that should either all be persisted together, or none should be persisted at all[14].

2.3.1 Shadow Paging. Shadow Paging allows for updates to be performed out-of-place. Instead of performing the update directly on the original object, it first creates a private copy of the object so that persistent updates can be applied to it without disturbing or modifying the original object. Since these private objects are local, they can be modified without worrying about the order of persist instructions. When a transaction is about to commit, the original objects are then replaced with the updated copies.

This approach presents two significant advantages in the context of PM. Private copies are stored in DRAM and are able to take advantage of its lower latency, allowing for better write performance, especially in the case of hot objects that are overwritten multiple times. A second advantage is that it can help improve the write endurance of PM systems by absorbing repeated overwrites in DRAM before issuing a single write operation to PM.

2.4 Durable HTM Solutions

In this section I illustrate an approach to durable HTM through the use of Shadow Paging by presenting SPHT[5], a state-of-the-art solution in durable HTM that combines Shadow Paging with Write-Ahead Logging and illustrates an approach used by a larger group of solutions, like DudeTM[2] and cc-HTM[3].

2.4.1 SPHT. SPHT, proposed by Daniel Castro et al. [5], builds on the concepts utilized in NV-HTM[1] and introduces novel mechanisms aimed at improving scalability during transaction processing and recovery. It addresses scalability challenges connected with redo logging by introducing a new highly scalable commit protocol that amortizes the cost of ensuring immediate durability[15] across multiple concurrent transactions.

The architecture for SPHT is comprised of 2 main processes:

- The transaction executer, which is comparable to the working process from NV-HTM. It spawns multiple worker threads responsible for executing the transactions and creates a shadow copy of the persistent heap shared by all threads and serves as a working snapshot that transactions access directly. Updates performed by HTM transactions on the working snapshot are not immediately written to the persistent heap and are thus still volatile.
- The log replayer, which spawns the replayer threads responsible for replaying the durably committed logs and updating the persistent heap.

Similarly to NV-HTM, each worker thread has its own private durable redo log used to track updates performed by each transaction. Since the results of a transaction can remain in the cache, the redo log needs to be explicitly flushed to persistent memory after the HTM commit. Once the redo log has been persisted, a timestamped commit marker is used to mark the transaction as durable.

SPHT takes advantage of the observation that at a high thread count, multiple transactions are likely to be concurrently attempting to commit. It takes advantage of this by ensuring the immediate durability of all transactions that are trying to commit through a single update of the persistent global marker with the timestamp of the most recent durable transaction. Just like NV-HTM, SPHT uses physical clocks to establish the order of transactions.

After an HTM commit, SPHT allows threads to flush their logs out of order, without considering thread synchronization. However, those logs cannot be marked as durable yet since they may depend on logs from other threads that may not yet have been flushed. It overcomes this by ensuring that each thread transaction waits for all threads with earlier timestamps to finalize persisting their logs. During this waiting phase, it also determines which transaction in the commit phase has the highest timestamp. If there is a transaction with a higher timestamp, the current transaction avoids updating the global marker. Only the transaction with the highest timestamp updates the global marker, which reduces the number of updates and flushes to the marker. This marks all transactions with earlier timestamps as durable.

SPHT also improves the scalability of log replay by employing 2 novel ideas, a log-linking mechanism that spares the replayer threads from the cost of having to determine which transaction should be replayed next, and parallelization of the log replay process in a Non-Uniform Memory Access (NUMA) aware fashion.

3 SPHT-EADR

The study of the state of the art presented in Section 2 shows that ensuring persistence in an ADR environment requires complex software instrumentation in order to guarantee that values do not remain lingering in volatile cache. The introduction of the new eADR domain in which caches can be considered durable means that it is now possible to implement durable HTM without the need for complex software instrumentation in order to identify which cache lines have been modified and guarantee they have been persisted. However, even though it is now possible to guarantee persistence entirely through HTM without the need for any additional software instrumentation, PM modules still suffer from finite write endurance and higher access latencies, particularly regarding write operations. Thus, it can still be beneficial to use some of these software techniques, such as shadow paging, in order to improve performance by taking advantage of the lower latency and higher performance of DRAM.

3.1 Study of Current Off-the-Shelf Approaches

The introduction of eADR leaves open the question of how to develop applications that are able to take advantage of this new persistency domain. Given that caches can now be considered persistent, one possible approach would be to avoid the use of software instrumentation techniques and rely entirely on existing HTM mechanisms. Another possible approach would be to use an existing state-of-the-art solution like SPHT[5], which was designed with ADR in mind, but can also be used in an eADR environment.

3.1.1 Pure HTM Approach. A pure HTM approach has the advantage of the simplicity of implementation, operating directly on the data stored in PM. It relies on mechanisms provided by the hardware without the added complexity of software instrumentation to ensure data has been persisted. However, even though it is now possible to guarantee the persistence of a successful transaction entirely through mechanisms provided by the hardware, it is still not possible to guarantee progress by relying exclusively on these mechanisms.

HTM is still a best-effort synchronization mechanism[12; 13] and requires a software-based fallback path in order to ensure progress. This fallback path can be implemented in the form of a Single Global Lock (SGL) that is used whenever a transaction repeatedly aborts over a pre-configured number of times. The SGL causes any concurrent transactions to be aborted, ensuring there are no conflicts with other transactions. Since this mechanism is implemented via software and memory access is performed directly on PM, any write operations made inside the SGL fallback path are written directly to PM. If an SGL transaction fails or aborts after having performed any write operations, those partial changes will still be present in PM, violating the principle of atomicity and leading to an incorrect system state. This requires the use of additional instrumentation to maintain consistency.

A possible solution for this problem when operating directly on PM is the use of an undo log used to track write operations

Algorithm 1 SGL with Undo Log

```
   Persistent Variables
1: pundoLog[]
   Thread Local Volatile Variables
2: vSGL

3: function beginTx
4:     while !CAS(&vSGL, 0, 1) do                ▷ Take SGL
5:         waitPrecedingTXs                      ▷ Writing the SGL causes an HTM abort
6:     createUndoLog                             ▷ Creates a new undo log
7: function write(addr, val)
8:     prevVal← *addr                            ▷ Save previous value
9:     undoLogWrite(addr, prevVal)               ▷ Log previous value PM
10:    *addr← val                                ▷ Execute write
11: function commitTx
12:    clearUndoLog                              ▷ Undo log no longer necessary
13:    SGL← 0                                    ▷ Release SGL, needs memory barrier
```

performed by an SGL transaction. In case of a failure or an aborted transaction, this undo log can be used to revert changes that had already been written to persistent memory and restore the consistent state of the system before that SGL transaction had been initiated. The algorithm for this is shown in Algorithm 1. As far as it was possible to determine, this thesis is the first work that implements and evaluates HTM for eADR using an instrumented fallback path.

3.1.2 Using SPHT. Even though SPHT was designed for an ADR environment and doesn't take advantage of the durable caches introduced with eADR, it can still be used in this new environment. All the properties SPHT guarantees in ADR are also enforced in the eADR environment. This approach is more complex and does not provide any benefits over executing in an ADR environment, but can potentially provide better performance due to the use of shadow paging in DRAM.

3.1.3 Comparative Study. A small comparative study of both these approaches was conducted in order to determine the feasibility of each solution and to determine if existing state-of-the-art solutions can still provide any benefits over the simpler approach of using pure HTM mechanisms with a software fallback path to ensure progress. The test for the study was performed with a synthetic benchmark in which each transaction generates a total of 5 read and 5 write operations at random over a uniform persistent heap.

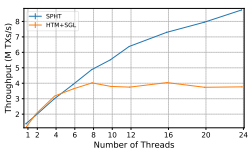

Fig. 1. Throughput comparison of SPHT and HTM+SGL with mixed access pattern

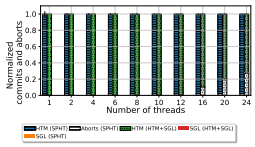

Fig. 2. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for SPHT and HTM+SGL with mixed access pattern

This study was conducted on an Intel Xeon processor with 12 cores and 24 threads; more detailed specifications for the machine

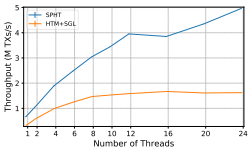

Fig. 3. Throughput comparison of SPHT and HTM+SGL with read-intensive access pattern

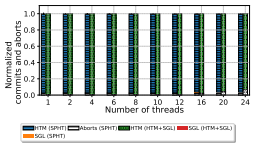

Fig. 4. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for SPHT and HTM+SGL with read-intensive access pattern

are available in Section 4.1. The tests were performed with a synthetic benchmark in which each transaction generates a fixed number of read and write operations at random over a uniform 1 GB persistent heap space in which each thread accesses its own private memory pool and all results are the average of 10 runs of each test.

In a first scenario, shown on Figure 1 and Figure 2, the workload is comprised of small transactions with a mixed access pattern in which each transaction generates a total of 5 read and 5 write operations. These results show that even though SPHT does not take advantage of the durable caches offered by the eADR environment, it still provides a significant advantage in throughput and scalability over the HTM approach, which is limited by the bandwidth capacity of the PM module.

In a second scenario, shown on Figure 3 and Figure 4, the workload is comprised of large transactions with a read-intensive access pattern in which each transaction generates a total of 45 read and 5 write operations. Latency times for write operations are significantly higher than for read operations in currently available PM modules[7]. A read-intensive access pattern could help avoid performance degradation in the HTM+SGL solution, but the results show that SPHT still provided a significant throughput and scalability advantage over HTM+SGL operating directly on PM.

These results show that the software instrumentation techniques used in state-of-the-art solutions like SPHT still provide benefits over operating directly on PM and motivate the need to develop a new solution that improves on existing state-of-the-art solutions by taking advantage of the new possibilities introduced with eADR.

3.2 Overview of SPHT

As mentioned previously, SPHT was originally developed for an ADR environment in which caches are considered volatile. This required the mechanisms described below, which can be seen in Algorithm 2, in order to ensure immediate durability and visibility of changes across threads.

3.2.1 Log Commit Marker. Each worker thread in SPHT has a private durable redo log that is used to log updates performed by each transaction. However, since caches are volatile, the updates performed by a transaction commit may still be lingering in volatile cache and not considered durable. This requires explicit flushing of the redo log to PM after the HTM commit terminates successfully. Since the log is only persisted after the transaction commits, SPHT

makes use of a durable timestamped log commit marker to indicate the transaction is considered durable.

3.2.2 Wait for Preceding Transactions. One of the key ideas used in SPHT in order to overcome scalability limitations is to amortize the cost of ensuring immediate durability across multiple transaction commits. When multiple transactions are concurrently trying to commit, SPHT is able to ensure immediate durability for all of them through a single update of the durable log commit marker by writing the timestamp of the most recent durable transaction.

However, since SPHT allows threads to flush logs out of order, flushing the transaction log for a given thread is not enough for that transaction to be considered durable. At that point, there may still exist preceding transactions with lower timestamps that are not yet marked as durable, but whose changes may already have been observed by other threads. In order to solve this issue, each thread shares the timestamp of the most recent transaction along with whether the logs for that transaction have been persisted. Once the logs have been flushed the transaction enters a phase in which it scans the timestamps of the other threads and waits until all transactions with lower timestamps have been persisted. Only once all transactions with lower timestamps have finished flushing their logs can it be considered safe for a transaction to mark itself as durable.

It is also during this waiting phase that threads identify which transaction has the most recent timestamp and will be responsible for updating and flushing the log commit marker.

3.2.3 Flush Transaction Log. When flushing cached logs to persistent memory, SPHT needs to perform calculations to determine if it is safe to flush the cache without the possibility of generating partial log writes to PM. If the full log and commit marker fit in a single cache line, it is safe to flush that cache line. However, if they occupy more than a single cache line, it is necessary for SPHT to flush the earlier cache lines and ensure the consistency of cache pages before flushing the commit marker to persistent memory.

3.3 SPHT Simplified for eADR

The introduction of durable caches in eADR has significant implications for the durability of redo logs, which was the primary motivator behind the previously described mechanisms. Since logs can now be considered durable in cache, it's no longer necessary to ensure they have been flushed to persistent memory before the transaction itself can be considered durable. The transaction can now be considered durable as soon as it commits successfully, rendering the global log marker unnecessary.

Likewise, it is now possible to consider preceding transactions with an earlier timestamp to be durable without the need to share whether their logs have been flushed to persistent memory or not. Any transaction with an earlier timestamp will have successfully committed and written their logs either to persistent memory or to cache, which is now considered durable. This means it is no longer necessary for transactions with an earlier timestamp to wait for any preceding transactions.

Algorithm 2 Original SPHT

```
   Shared Volatile Variables
1: ᵛts[N], ᵛmarked[N], ᵛisUpd[N]
   Persistent Variables
2: ᴾwriteLog[N], ᴾmarker
   Thread Local Volatile Variables
3: ᵛts′, ᵛskipCAS

4: function beginTx
5:     ᵛisUpd[myTid] ← false
6:     ᵛskipCAS ← false
7:     unsetPersBit(ᵛts[myTid])                    ▷ Logs are not persistent
8:     ᵛts[myTid] ← rdtscp                         ▷ lower bound of final ts
9:     htm_begin                                   ▷ begin hw tx
10: function write(addr, val)
11:    logWrite(addr, val)                         ▷ log to PM, no flush
12:    *addr← val                                  ▷ execute write
13: function commitTx
14:    ᵛts′ ← rdtscp                               ▷ store physical clock to local var.
15:    htm_commit                                  ▷ commit hw transaction
16:    ᵛts[myTid] ← ts′                            ▷ Externalize the final timestamp
17:    if isReadOnly then                          ▷ Read-only txs...
18:        setPersBit(ᵛts[myTid])                  ▷ ...unblock the others
19:        return                                  ▷ ...and return immediately
20:    ᵛisUpd[myTid] ← true                        ▷ Mark as update tx
21:    logCommit(ᴾwriteLog[myTid], ts′)            ▷ Flush tx log.
22:    setPersBit(ᵛts[myTid])                      ▷ Signal logs are durable
23:    waitPrecedingTXs
24:    updateMarker
25: function waitPrecedingTXs
26:    for t ∈ [0..N − 1] do
27:                                                ▷ Wait until prec. txs have flushed their logs
28:        while ᵛts[t] < ᵛts[myTid] ∧ ¬isPersBit(ᵛts[t]) wait
29:                                                ▷ If any update tx with large ts exists...
30:        if ᵛts[t] > ᵛts[myTid]∧ ᵛisUpd[t] then
31:            ᵛskipCAS ← true                     ▷ this tx can skip the CAS
32: function updateMarker
33:                                                ▷ Is it needed to and am I responsible for updating ᴾmarker?
34:    if ᴾmarker < ᵛts[myTid] ∧ ¬ᵛskipCAS then
35:        val← ᴾmarker
36:        while val < ᵛts[myTid] do
37:            val ← CAS(ᴾmarker, val, ᵛts[myTid])
38:        if (CAS was successful) then
39:            flush(ᴾmarker)
40:            ᵛmarked[myTid] ← ᵛts[myTid]         ▷ Signals ᴾmarker is flushed.
41:            return
42:    while true do                               ▷ Wait till flush of ᴾmarker
43:        for t ∈ [0..N − 1] do                   ▷ ...is complete
44:            if ᵛmarked[t] ≥ ᵛts[myTid] then return
```

Additionally, the process of flushing logs to persistent memory, along with determining which cache lines need to be flushed, is no longer necessary.

These changes allow the algorithm for SPHT-eADR to take advantage of the new possibilities introduced in eADR and to be significantly simplified in comparison to the original version of SPHT, as shown in Algorithm 3.

3.3.1 Maintaining Flushes. Even though flush operations are no longer required to ensure the persistence in an eADR environment, they may still be beneficial for performance by proactively removing cache lines containing data which no longer has temporal locality[16], as is the case with log entries for transactions that have already been committed. This is shown on Line 17 of Algorithm 4

Given that changes lingering in cache can be considered persistent and that the version of SPHT optimized for eADR no longer needs to maintain a commit marker, it is possible to flush redo logs using an out-of-order operation like CLWB or CLFLUSHOPT without the need to issue a memory fence and wait for the flushes to finish.

Algorithm 3 SPHT-eADR

```
   Persistent Variables
1: ᴾwriteLog[N]
   Thread Local Volatile Variables
2: ᵛisUpdate
3: function BEGINTX
4:     HTM_BEGIN                                          ▷ Begin hw tx
5: function WRITE(addr, val)
6:     ᵛisUpdate ← true
7:     logWrite(addr, val)                                ▷ Log to PM, no flush
8:     *addr← val                                         ▷ Execute write
9: function COMMITTX
10:    if isUpdate then
11:        logCommit(ᴾwriteLog[myTid], RDTSCP)            ▷ No flush required
12:    HTM_COMMIT                                         ▷ SGL commit needs a memory barrier
```

Algorithm 4 SPHT-eADR with Flushes

```
   Persistent Variables
1: ᴾwriteLog[N]
   Thread Local Volatile Variables
2: ᵛisUpdate, ᵛtxLogStart, ᵛtxLogEnd
3: function BEGINTX
4:     ᵛtxLogStart ← logNextPos()                         ▷ Record starting log position
5:     ᵛtxLogEnd ← txLogStart
6:     HTM_BEGIN                                          ▷ Begin hw tx
7:     *addr← val                                         ▷ Execute write
8: function WRITE(addr, val)
9:     ᵛisUpdate ← true
10:    logWrite(addr, val)                                ▷ Log to PM, no flush
11:    ᵛtxLogEnd ← logNextPos()                           ▷ Update current log position
12:    *addr← val                                         ▷ Execute write
13: function COMMITTX
14:    if isUpdate then
15:        logCommit(ᴾwriteLog[myTid], RDTSCP)
16:    HTM_COMMIT                                         ▷ SGL commit needs a memory barrier
17:    flushCache(txLogStart, txLogEnd)                   ▷ Flush cache for updated log section
```

3.4 Implementation of SPHT-eADR

SPHT-eADR exposes PM to the application via a persistent heap created by memory-mapping the persistent data stored in a PM-aware filesystem into the application address space[17] using the host Operating System (OS). Being based on SPHT, SPHT-eADR follows the same architecture with two main processes (Transaction Executers and Log Replayers) that were earlier described in Section 2.4.1. The transaction executor process memory-maps a persistent heap into its address space using Copy-on-Write provided by the OS which creates a shadow copy of the persistent heap. The process also spawns multiple worker threads that share access to this shadow copy. Changes to the shadow copy are not transmitted back to the persistent heap. Instead, worker threads track updates through private redo logs, implemented with a circular buffer, which contains an ordered sequence of transactions and timestamps. These logs can eventually be replayed in order to propagate changes back to the persistent heap. Transactions in SPHT-eADR utilize the underlying support for HTM and switch to a fallback Single Global Lock software-based commit mechanism when a transaction fails a pre-configured number of times. When this fallback mode is activated, all concurrent hardware-based transactions are immediately aborted.

4 EXPERIMENTAL EVALUATION

This chapter presents the results of an experimental evaluation of SPHT-eADR, a new solution based on SPHT and optimized for an eADR environment (previously described in Section 3.3), and seeks to answer the question of whether shadow paging and software instrumentation techniques used by state-of-the-art solutions like SPHT can still be used to improve performance given the availability of eADR. The performance of SPHT-eADR was compared to the standard version of SPHT and an almost pure HTM mechanism with a software fallback path to ensure progress, as well as different versions of SPHT-eADR with preemptive flushing of logs with low temporal locality. These solutions were evaluated using synthetic benchmarks with no contention, STAMP, and TPC-C.

4.1 Experimental Settings

All experiments were conducted in a dual-socket system using a single Intel Xeon Gold 5317 3.00 GHz 3rd Generation Intel Xeon Scalable processor with 12 cores and 24 hardware threads, equipped with 128 GB of DRAM (4x 32 GB) and 512 GB of Intel Optane DC PM 200 (4x 128 GB) with interleaved access and configured in App Direct mode. These experiments evaluate the performance of:

- HTM+SGL: plain HTM with a software fallback using a single global lock;
- SPHT[5]: original version of SPHT developed for an ADR environment;
- SPHT-eADR: new solution based on SPHT and optimized for an eADR environment; see Section 3.3;
- SPHT-eADR-Flush: a version of SPHT-eADR with logic to flush the redo log after a successful commit; see Section 3.3.1.

All described solutions make use of HTM and fall back to SGL when a transaction fails after 10 retries. All results are the average of 10 runs.

The synthetic benchmark is configured with a 1 GB heap space which is split into private memory pools for each thread, ensuring that there are no conflicting transactions. Before beginning the execution of the benchmark, memory pages are pre-touched in order to simulate a long-running process. Each iteration of the test runs for 5 seconds and each transaction generates a pre-configured number of read and write operations to random memory addresses within the memory pool for each thread. This benchmark evaluates the performance of the various solutions in scenarios where every transaction is able to be executed concurrently without incurring conflicts in order to evaluate the scalability and possible bottlenecks for each solution.

STAMP[18] is a benchmark suite designed for transactional memory systems and includes transactional applications that, even though they were not originally designed with PM in mind, could still be able to benefit from crash-tolerance in a PM system. STAMP was previously used to test SPHT, along with several other related solutions[1; 3; 4; 19] in the same field.

Although the STAMP benchmark suite includes 8 different benchmarks, this evaluation does not consider the Bayes application, as it is known to provide unstable performance results[5; 20].

TPC-C[19] is a well-known benchmark that is widely used to evaluate database and transactional systems. The benchmark is

composed of five transactions: three update transactions (New Order, Payment, and Delivery), and two read-only transactions (Status Order and Stock Level). This evaluation implemented three of these transactions, Payment, New Order, and Delivery. New Order and Delivery are transactions that contain item accesses dependent on other previous accesses.

4.2 Flushing Approach

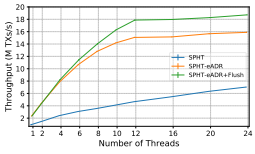

Fig. 5. Throughput for synthetic benchmark with 5 writes and 5 reads using CLFLUSHOPT

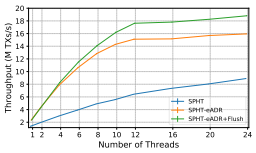

Fig. 6. Throughput for synthetic benchmark with 5 writes and 5 reads using CLWB

This synthetic benchmark experiment compares the throughput and breakdown of committed and aborted transactions of the regular version of SPHT, SPHT-eADR without flushes, and SPHT-eADR with flushes using a balanced workload in which each transaction performs 5 read and 5 write operations. The test was repeated using both CLWB and CLFLUSHOPT operations in order to implement the log flushing phase taking place after transaction commit in these solutions. The CLFLUSHOPT instruction flushes data out of the CPU cache and invalidates it whereas CLWB flushes the data without invalidating the cache lines. This allows for evaluating not just the impact of preemptively flushing logs to PM but also determining whether cache invalidation has any negative effect. Figure 5 and Figure 6 show the throughput of the 3 solutions using CLFLUSHOPT and CLWB, respectively. SPHT-eADR and SPHT-eADR with flushes have similar throughput curves, scaling well up to the number of physical cores. Once Hyper-Threading is used, the curve flattens and throughput stays almost constant. The version with flushes performs noticeably better than the version without flushes, indicating that preemptively flushing data with low temporal locality does provide a performance boost. We argue that this increases the effectiveness of the caching layer by asking the hardware to flush log data that is unlikely to be reaccessed shortly thereafter. The original version of SPHT scales more linearly up to 24 threads but at much lower throughput levels. The results of the tests with CLWB and CLFLUSHOPT are similar, indicating that invalidating the cache lines for the redo logs, which have low temporal locality, does not have a negative effect on performance.

Figure 7 and Figure 8 show the breakdown of committed and aborted transactions for the solutions, with committed transactions being split into HTM or SGL commit mechanisms. With this workload, the abort rate is very low, only growing a bit at higher thread counts. However, it is worth noting that even though the throughput of the original version of SPHT is lower, it does have a lower abort rate than SPHT-eADR and SPHT-eADR with flushes. The higher throughput of SPHT-eADR means that more write requests reach the write-pending queue of the PM module, generating more aborts.

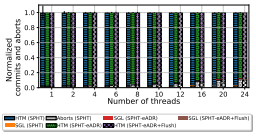

Fig. 7. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 5 writes and 5 reads using CLFLUSHOPT

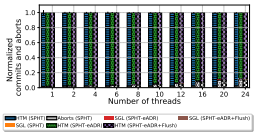

Fig. 8. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 5 writes and 5 reads using CLWB

4.3 Evaluating SPHT-eADR

4.3.1 Synthetic Benchmark. This experiment includes 6 variations of the synthetic (no contention) benchmark, covering a variety of scenarios encompassing all combinations of small or large transactions (10 or 50 memory accesses) with read-intensive, write-intensive, or mixed access patterns. The lack of conflicts in this case also allows for the evaluation of scalability and identification of possible bottlenecks for each solution.

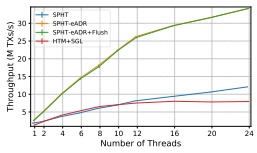

Fig. 9. Throughput for synthetic benchmark with 1 write and 9 reads

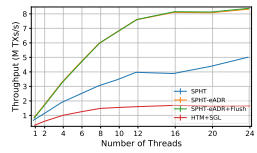

Fig. 10. Throughput for synthetic benchmark with 5 writes and 45 reads

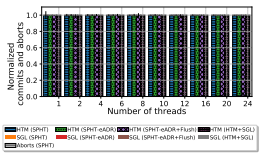

Fig. 11. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 1 write and 9 reads

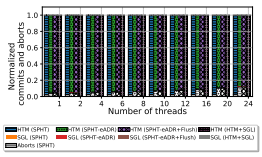

Fig. 12. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 5 writes and 45 reads

SPHT-eADR has the best performance in read-intensive applications (see Figure 9 and Figure 10), reaching 2–3x the throughput performance of SPHT and 4x the throughput of HTM+SGL. SPHT-eADR with flush operations performs very close to SPHT-eADR, but flushing does not present an advantage in these applications. All versions of SPHT only generate log entries for write operations. As such, it is expectable for flushing the logs to have a small impact here given the small amount of memory generated by logs in cache.

With a mixed access pattern (see Figure 13 and Figure 14) applications start to give SPHT-eADR with flushing an advantage,

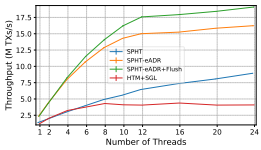

Fig. 13. Throughput for synthetic benchmark with 5 writes and 5 reads

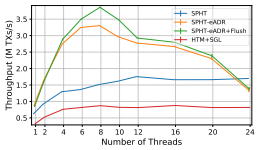

Fig. 14. Throughput for synthetic benchmark with 25 writes and 25 reads

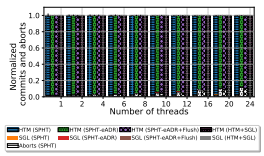

Fig. 15. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 5 writes and 5 reads

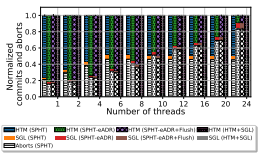

Fig. 16. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 25 writes and 25 reads

performing better than all other solutions. Write operations are noticeably slower though, causing the throughput for each application to reduce significantly compared to the read-intensive application performing the same number of operations. Again, SPHT-eADR and SPHT-eADR with flushes reach roughly 2x the performance of SPHT and 3–4x that of HTM+SGL.

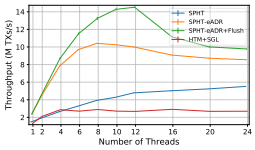

Fig. 17. Throughput for synthetic benchmark with 9 writes and 1 read

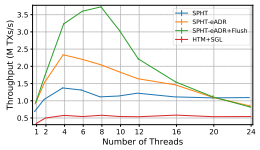

Fig. 18. Throughput for synthetic benchmark with 45 writes and 5 reads

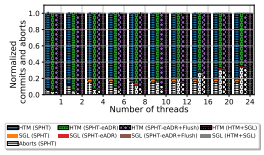

Fig. 19. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 9 writes and 1 read

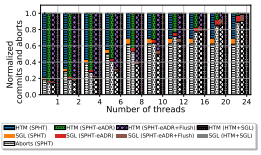

Fig. 20. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 45 writes and 5 reads

In contrast with read-intensive applications, write-intensive applications give SPHT-eADR with flushes a significant advantage in performance, albeit mostly at lower thread counts. Throughput for SPHT-eADR peaks at 8–12 threads, and declines noticeably after that, particularly in applications with a combination of large transactions and write-intensive access patterns.

The original version of SPHT has lower peak throughput but does not suffer from degraded performance with a high number of threads due to the waiting mechanism used in the commit phase which helps maintain a lower number of aborted transactions at higher thread counts.

Overall, taking these results as a whole, it is possible to see that HTM+SGL reaches a plateau early on with just a few threads and does not scale further, being limited by the higher latency of PM when compared to DRAM. The new versions of SPHT-eADR and SPHT-eADR with flushes scale better and reach much higher peak throughput with a lower number of cores. However, performance degrades with a higher number of threads, especially for large and write-intensive workloads.

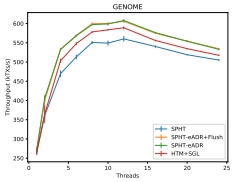

Fig. 21. Throughput for GENOME benchmark

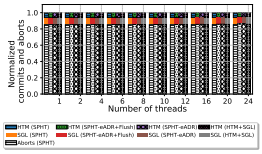

Fig. 22. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for GENOME benchmark

4.3.2 STAMP Benchmark. GENOME (see Figure 21) is a medium contention benchmark and not very favourable towards scalability, given that there is a high likelihood of generating conflicts between transactions, as can be seen on Figure 22 by the abort rate of over 80%. SPHT-eADR (with and without flushes) performed the best in this benchmark and reached peak throughput at 12 threads. The results for all solutions are quite closely correlated, which may be due to the high number of aborts causing most transactions to be committed via the SGL.

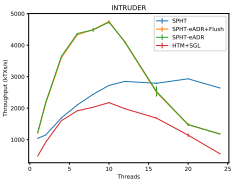

Fig. 23. Throughput for INTRUDER benchmark

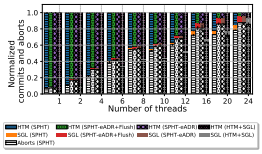

Fig. 24. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for INTRUDER benchmark

INTRUDER (see Figure 23) is also a contention-prone benchmark that is not favourable to scalability. SPHT-eADR (with and without flushes) reaches 1.5–2x higher peak throughput than other solutions, but degrades quickly as the number of threads increases due to a

corresponding increase in the number of aborts, as can be seen on Figure 24. The original version of SPHT stays significantly lower in terms of maximum throughput but scales more gracefully to a large number of threads without degrading performance.

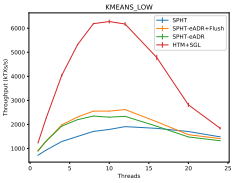

Fig. 25. Throughput for KMEANS LOW benchmark

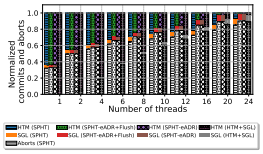

Fig. 26. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for KMEANS LOW benchmark

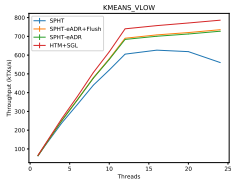

Fig. 27. Throughput for KMEANS VLOW benchmark

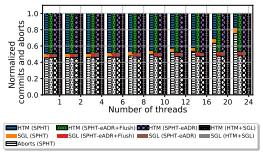

Fig. 28. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for KMEANS VLOW benchmark

HTM+SGL performs very favourably with KMEANS LOW (see Figure 25) and KMEANS VLOW (see Figure 27), both in terms of throughput and abort rate, achieving over 2x the throughput in KMEANS LOW when compared to all the versions of SPHT and SPHT-eADR. This can be explained by the fact that HTM+SGL has a much lower abort rate at lower thread counts and is able to commit via the HTM path most of the times. In KMEANS VLOW the slowdown is not as large as in KMEANS LOW, with all solutions reaching their peak throughput at 12 threads and stabilizing when additional HyperThreading threads are added. Looking at the committed and aborted transaction breakdown in Figure 26 and Figure 28 it is possible to see that the number of aborted transactions increases with the number of threads in KMEANS LOW and stays stable in KMEANS VLOW, with the exception of the original version of SPHT that starts getting more aborted transactions at high thread counts. HTM+SGL is able to maintain a much lower abort rate in this case.

Both SSCA2 and VACATION LOW (see Figure 29 and Figure 31) are low-contention benchmarks that are favourable to HTM. This can be seen on Figure 30 which shows that SSCA2 has a 100% commit rate entirely through HTM, without falling back to SGL. All tested solutions scale smoothly up to 24 threads, but these are the benchmarks where both versions of SPHT-eADR show the best scalability, reaching 2x the peak throughput of HTM+SGL and 1.2x of SPHT without any degradation in performance at higher thread counts.

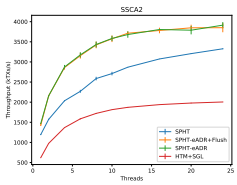

Fig. 29. Throughput for SSCA2 benchmark

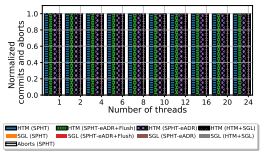

Fig. 30. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for SSCA2 benchmark

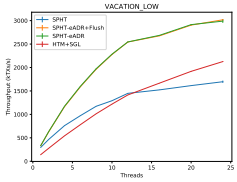

Fig. 31. Throughput for VACATION LOW benchmark

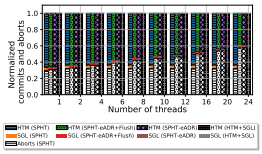

Fig. 32. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for VACATION LOW benchmark

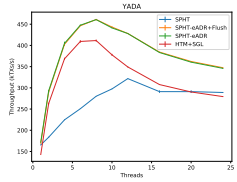

Fig. 33. Throughput for YADA Benchmark

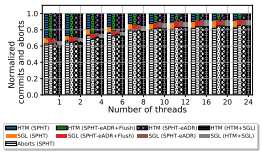

Fig. 34. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for YADA benchmark

YADA (see Figure 33) is another contention-prone benchmark that generates large transactions, providing an unfavourable running environment for HTM with a high percentage of aborted transactions, visible in Figure 34. Both versions of SPHT-eADR perform significantly better than SPHT, but peak throughput is reached at around 8 threads and performance degrades significantly after that. This is to be expected from a contention-prone benchmark like YADA.

4.3.3 TPC-C Benchmark. Figure 35 and Figure 36 show the results of TPC-C implemented with the three update transactions: New Order, Payment, and Delivery and configured with 32 warehouses, 95% payment, 3% delivery, and 2% new order transactions. Both versions of SPHT-eADR, with and without flushes, perform very similarly, scaling very well up to 12 threads with 1.5x the throughput of SPHT but degrading rapidly from thread 13 on, once HyperThreading is in use. The original version of SPHT scales very favourably in this test, however. It does not reach the same maximum throughput that SPHT-eADR is able to reach at 12 threads, but it continues scaling upwards even with HyperThreading due to the waiting mechanism used in the commit phase. Figure 36 shows that the number of aborted transactions increases with the number of threads for most

solutions, with the exception of SPHT which generates a lower percentage of aborted transactions with higher thread counts.

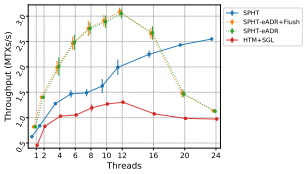

Fig. 35. Throughput for TPC-C

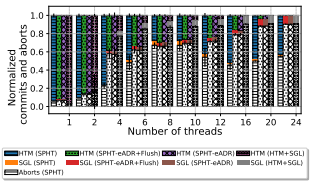

Fig. 36. Breakdown of committed (via the HTM and SGL paths) and aborted transactions for TPC-C

5 CONCLUSION

Having revisited and studied these proposals, this dissertation introduced SPHT-eADR, a new solution for durable HTM optimized for systems with durable caches that makes use of DRAM shadow paging techniques.

This approach had not yet been studied in an eADR environment, but the experimental evaluation of SPHT-eADR shows that it significantly improves on the performance of previous state-of-the-art solutions by providing higher performance and fewer overheads, reaching 2–3x the throughput performance of SPHT and 4x the throughput of HTM+SGL in synthetic (no contention) benchmarks and 1.5x the throughput of SPHT and 2x the throughput of HTM+SGL at similar thread counts on TPC-C.

5.1 Future Work

One topic that was approached during the execution of this dissertation was the issue of support for large heap allocation in systems that make use of DRAM shadow paging. Most state-of-the-art systems are limited by the size of the DRAM pool available, causing them to either fail or drastically degrade performance once that limit is reached due to the cost of the operating system swapping memory in and out to disk. The main strategy that has been considered in the literature and is utilized by DudeTM[2] consists of paying the cost for restoring the content on page-in, when a page fault occurs. However, there are unexplored regions and alternative approaches that seem interesting and can potentially improve performance, such as shifting the cost to the page-out event instead, avoiding overheads on page-in events during execution.

Another future avenue of research would be the prevention of performance degradation at higher thread counts, which could be addressed with the introduction of rate-limiting or some other form of back-off mechanism. One interesting path would be how to automatically determine the amount of rate-limiting or back-off time required to prevent performance degradation at higher thread counts without hurting performance at lower thread counts, borrowing ideas from previous literature in the area of self-tuning[21; 22].

REFERENCES

[1] D. Castro, P. Romano, and J. Barreto, "Hardware Transactional Memory Meets Memory Persistency," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 368–377.

[2] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building Durable Transactions with Decoupling for Persistent Memorya," in *ASPLOS '17: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2017, pp. 329––343.

[3] E. Giles, K. Doshi, and P. Varman, "Continuous checkpointing of htm transactions in nvm," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. Association for Computing Machinery, 2017, p. 70–81.

[4] K. Genç, M. D. Bond, and G. H. Xu, "Crafty: efficient, HTM-compatible persistent transactions," in *PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2020, pp. 59–74.

[5] D. Castro, A. Baldassin, J. Barreto, and P. Romano, "SPHT: Scalable Persistent Hardware Transactions," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 155–169.

[6] P. Zardoshti, M. Spear, A. Vosoughi, and G. Swart, "Understanding and Improving Persistent Transactions on Optane™ DC Memory," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 348–357.

[7] Intel, "Affordably Accommodate the Next Wave of Data Demands," https://www.intel.com/content/dam/www/public/us/en/documents/brief/affordably-accommodate-the-next-wave-of-data-demands.pdf, Last accessed 21 May 2021.

[8] ——, "Achieve Greater Insight from Your Data with Intel Optane Persistent Memory," https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-persistent-memory-200-series-brief.pdf, Last accessed 21 May 2021.

[9] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 4th ed. McGraw-Hill Education - Europe, 2002.

[10] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, January 2010, pp. 47–56.

[11] Nuno Diegues and Paolo Romano and Luís Rodigues, "Virtues and Limitations of Commodity Hardware Transactional Memory," in *PACT '14: Proceedings of the 23rd international conference on Parallel architectures and compilation*, August 2017, pp. 3–14.

[12] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf, Last accessed 21 May 2021.

[13] B. Hall, P. Bergner, A. S. Housfater, M. Kandasamy, T. Magno, A. Mericas, S. Munroe, M. Oliveira, B. Schmidt, W. Schmidt, B. K. Smith, J. Wang, S. Warrier, and D. Wendt, "Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8," http://www.redbooks.ibm.com/redbooks/pdfs/sg248171.pdf, Last accessed 21 May 2021.

[14] A. Baldassin, J. a. Barreto, D. Castro, and P. Romano, "Persistent memory: A survey of programming support and implementations," *ACM Comput. Surv.*, vol. 54, no. 7, jul 2021. [Online]. Available: https://doi.org/10.1145/3465402

[15] R. M. Krishnan, J. Kim, A. Mathew, X. wei Fu, A. Demeri, C. Min, and S. sun Kannan, "Durable transactional memory can scale with TimeStone," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 335–349.

[16] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proc. VLDB Endow.*, vol. 14, no. 4, p. 626–639, dec 2020. [Online]. Available: https://doi.org/10.14778/3436905.3436921

[17] Storage Networking Industry Association (SNIA) Technical Position, "NVM Programming Model Version 1.2," jun 2017.

[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[19] Transaction Processing Performance Council, "TPC-C Benchmark Revision 5.11.0," http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, Last accessed 21 May 2021.

[20] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, "Evaluation of amd's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 27–40. [Online]. Available: https://doi.org/10.1145/1755913.1755918

[21] N. Diegues and P. Romano, "Self-tuning intel restricted transactional memory," *Parallel Computing*, vol. 50, pp. 25–52, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819115001209

[22] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," *Computing*, vol. 97, no. 9, p. 939–959, sep 2015. [Online]. Available: https://doi.org/10.1007/s00607-013-0376-3